

**Department Of Computer Engineering**

**Microprocessor**

**Lab Experiments**

SUBMITTED TO THE DEPARTMENT OF COMPUTER ENGINEERING

AISSMS IOIT

**SE COMPTER ENGINEERING**

**SUBMITTED BY**

**Ashish Patil**

**ERP NO**

**–**

**51**

**2020**

**-**

**2021**

**Home**

# EXPERIMENT NO. 01

**NAME:** Content to bridge the GAP- Write an X86/64 ALP to display Hello World using macros.

**AIM: :** Write an X86/64 ALP to display Hello World using Macros.

**OBJECTIVES:**

* To understand assembly language programming instruction set
* To understand different assembler directives with example
* To apply instruction set for implementing X86/64 bit assembly language programs

**ENVIRONMENT:**

* Operating System: 64-bit Open source Linux or its derivative.
* Programming Tools: Preferably using Linux equivalent or MASM/TASM/NASM/FASM.  Text Editor: geditor

**THEORY:**

**Introduction to Assembly Language Programming:**

Each personal computer has a microprocessor that manages the computer's arithmetical, logical and control activities. Each family of processors has its own set of instructions for handling various operations like getting input from keyboard, displaying information on screen and performing various other jobs. These set of instructions are called 'machine language instruction'. Processor understands only machine language instructions which are strings of 1s and 0s. However machine language is too obscure and complex for using in software development. So the low level assembly language is designed for a specific family of processors that represents various instructions in symbolic code and a more understandable form. Assembly language is a low-level programming language for a computer, or other programmable device specific to particular computer architecture in contrast to most high-level programming languages, which are generally portable across multiple systems. Assembly language is converted into executable machine code by a utility program referred to as an assembler like NASM, MASM etc.

**MACROS**:

Writing a macro is another way of ensuring modular programming in assembly language.  A macro is a sequence of instructions, assigned by a name and could be used anywhere in the program.

* In NASM, macros are defined with **%macro** and **%endmacro** directives.
* The macro begins with the %macro directive and ends with the %endmacro directive.

The Syntax for macro definition –

%macro macro\_name number\_of\_params

<macro body>

%endmacro

Where, *number\_of\_params* specifies the number parameters, *macro\_name* specifies the name of the macro.

The macro is invoked by using the macro name along with the necessary parameters. When you need to use some sequence of instructions many times in a program, you can put those instructions in a macro and use it instead of writing the instructions all the time.

For example, a very common need for programs is to write a string of characters in the screen. For displaying a string of characters, you need the following sequence of instructions –

|  |  |
| --- | --- |
| mov | edx,len ;message length |
| mov | ecx,msg ;message to write |
| mov | ebx,1 ;file descriptor (stdout) |
| mov | eax,4 ;system call number (sys\_write) |
| int | 0x80 ;call kernel |

In the above example of displaying a character string, the registers EAX, EBX, ECX and EDX have been used by the INT 80H function call. So, each time you need to display on screen, you need to save these registers on the stack, invoke INT 80H and then restore the original value of the registers from the stack. So, it could be useful to write two macros for saving and restoring data.

We have observed that, some instructions like IMUL, IDIV, INT, etc., need some of the information to be stored in some particular registers and even return values in some specific register(s). If the program was already using those registers for keeping important data, then the existing data from these registers should be saved in the stack and restored after the instruction is executed.

**ALGORITHM**:

Editing, Assembling, Linking and Executing First Program : hello.asm

1. Boot the machine with ubuntu
2. Select and click on <dash home> icon from the toolbar.
3. Start typing "terminal".
4. Click on "terminal" icon. A terminal window will open showing command prompt.
5. Give the following command at the prompt to invoke the editor gedit hello.asm

[3:55 PM, 6/6/2021] Akash Mete: 6. Type in the program in gedit window, save and exit 7. To assemble the program write the command at the prompt as follows and press enter key nasm -f elf32 hello.asm -o hello.o (for 32 bit) nasm -f elf64 hello.asm -o hello.o (for 64 bit)

1. If the execution is error free, it implies hello.o object file as been created.
2. To link and create the executable give the command as

Id -o hello hello.o

1. To execute the program write at the prompt

./hello

1. Hello, world will be displayed at the prompt.

**PROGRAM**:

ALP to Print "Hello World" using 64-bit model section data message: db 'Hello, world', Ox0A ; message and newline length: equ $-message

; length of message string section text global\_start

; global entry point export for ld start:

mov

rax, 1 mov rdi, 1 mov rsi, message mov rdx, length syscall

; 01 specifies sys write kemel call

; 01 specifies stdout

: Load starting address of message into rsi

; Load message string length into rdx mov

rax, 60 mov

; sys\_exit rdi, o ; return 0 (success)

Syscall

**FLOWCHART:**

**START**

Initialize data segment and

store message “Hello World

”

In text segment, jump to label message

and display

message.

**END**



**CONCLUSION:**

In this practical session we learnt how to display Hello World using macros.

# EXPERIMENT NO. 02

**NAME:** Write an X86/64 ALP to accept five 64 bit Hexadecimal numbers from user and store them in an array and display the accepted numbers.

**AIM: :** Write an X86/64 ALP to accept five 64 bit Hexadecimal numbers from user and store them in an array and display the accepted numbers.

**OBJECTIVES:**

* To understand assembly language programming instruction set
* To understand different assembler directives with example
* To apply instruction set for implementing X86/64 bit assembly language programs

**ENVIRONMENT:**

* Operating System: 64-bit Open source Linux or its derivative.
* Programming Tools: Preferably using Linux equivalent or MASM/TASM/NASM/FASM.  Text Editor: geditor

**THEORY:**

**Introduction to Assembly Language Programming:**

Each personal computer has a microprocessor that manages the computer's arithmetical, logical and control activities. Each family of processors has its own set of instructions for handling various operations like getting input from keyboard, displaying information on screen and performing various other jobs. These set of instructions are called 'machine language instruction'. Processor understands only machine language instructions which are strings of 1s and 0s. However machine language is too obscure and complex for using in software development. So the low level assembly language is designed for a specific family of processors that represents various instructions in symbolic code and a more understandable form. Assembly language is a low-level programming language for a computer, or other programmable device specific to particular computer architecture in contrast to most high-level programming languages, which are generally portable across multiple systems. Assembly language is converted into executable machine code by a utility program referred to as an assembler like NASM, MASM etc.

**Advantages of Assembly Language**

An understanding of assembly language provides knowledge of:

Interface of programs with OS, processor and BIOS;

Representation of data in memory and other external devices;

How processor accesses and executes instruction; How instructions accesses and process data; How a program access external devices.



Other advantages of using assembly language are:

It requires less memory and execution time;

It allows hardware-specific complex jobs in an easier way; It is suitable for time-critical jobs;



**ALP Step By Step:**

**Installing NASM:**

If you select "Development Tools" while installed Linux, you may NASM installed along with the Linux operating system and you do not need to download and install it separately. For checking whether you already have NASM installed, take the following steps:

Open a Linux terminal.

Type ***whereis nasm*** and press ENTER.

If it is already installed then a line like, *nasm: /usr/bin/nasm* appears. Otherwise, you will see just*nasm:*, then you need to install NASM.



**To install NASM take the following steps:**

Open Terminal and run below commands:

sudo apt-get update sudo apt-get install nasm

**Assembly Basic Syntax:**

An assembly program can be divided into three sections:

The **data** section

The **bss** section



The **text** section

The order in which these sections fall in your program really isn’t important, but by convention the .data section comes first, followed by the .bss section, and then the .text section.

**The .data Section**

The .data section contains data definitions of initialized data items. Initialized data is data that has a value before the program begins running. These values are part of the executable file. They are loaded into memory when the executable file is loaded into memory for execution. You don’t have to load them with their values, and no machine cycles are used in their creation beyond what it takes to load the program as a whole into memory. The important thing to remember about the .data section is that the more initialized data items you define, the larger the executable file will be, and the longer it will take to load it from disk into memory when you run it. **The .bss Section**

Not all data items need to have values before the program begins running. When you’re reading data from a disk file, for example, you need to have a place for the data to go after it comes in from disk. Data buffers like that are defined in the .bss section of your program. You set aside some number of bytes for a buffer and give the buffer a name, but you don’t say what values are to be present in the buffer. There’s a crucial difference between data items defined in the .data section and data items defined in the .bss section: data items in the .data section add to the size of your executable file. Data items in the .bss section do not.

**The .text Section**

The actual machine instructions that make up your program go into the .text section. Ordinarily, no data items are defined in .text. The .text section contains symbols called *labels* that identify locations in the program code for jumps and calls, but beyond your instruction mnemonics, that’s about it.

All global labels must be declared in the .text section, or the labels cannot be ‘‘seen’’ outside your program by the Linux linker or the Linux loader. Let’s look at the labels issue a little more closely.

**Labels**

A label is a sort of bookmark, describing a place in the program code and giving it a name that’s easier to remember than a naked memory address. Labels are used to indicate the places where jump instructions should jump to, and they give names to callable assembly language procedures.

Here are the most important things to know about labels:

 *Labels must begin with a letter, or else with an underscore, period, or question mark.* These last three have special meanings to the assembler, so don’t use them until you know how NASM interprets them.

 *Labels must be followed by a colon when they are defined.* This is basically what tells NASM that the identifier being defined is a label. NASM will punt if no colon is there and will not flag an error, but the colon nails it, and prevents a mistyped instruction mnemonic from being mistaken for a label. Use the colon!

 *Labels are case sensitive.* So yikes:, Yikes:, and YIKES: are three completely different labels.

**Assembly Language Statements**

Assembly language programs consist of three types of statements:

Executable instructions or instructions

Assembler directives or pseudo-ops Macros



**Syntax of Assembly Language Statements**

[label] mnemonic [operands] [;comment] **LIST OF INTERRRUPTS USED:** NA

**LIST OF ASSEMBLER DIRECTIVES USED:** EQU,DB

**LIST OF MACROS USED:** NA

**LIST OF PROCEDURES USED:** NA

**ALGORITHM:**

INPUT: ARRAY

OUTPUT: ARRAY

STEP 1: Start.

STEP 2: Initialize the data segment.

STEP 3: Display msg1 “Accept array from user. “

STEP 4: Initialize counter to 05 and rbx as 00

STEP 5: Store element in array.

STEP 6: Move rdx by 17.

STEP 7: Add 17 to rbx.

STEP 8: Decrement Counter.

STEP 9: Jump to step 5 until counter value is not zero.

STEP 9: Display msg2.

STEP 10: Initialize counter to 05 and rbx as 00

STEP 11: Display element of array.

STEP 12: Move rdx by 17.

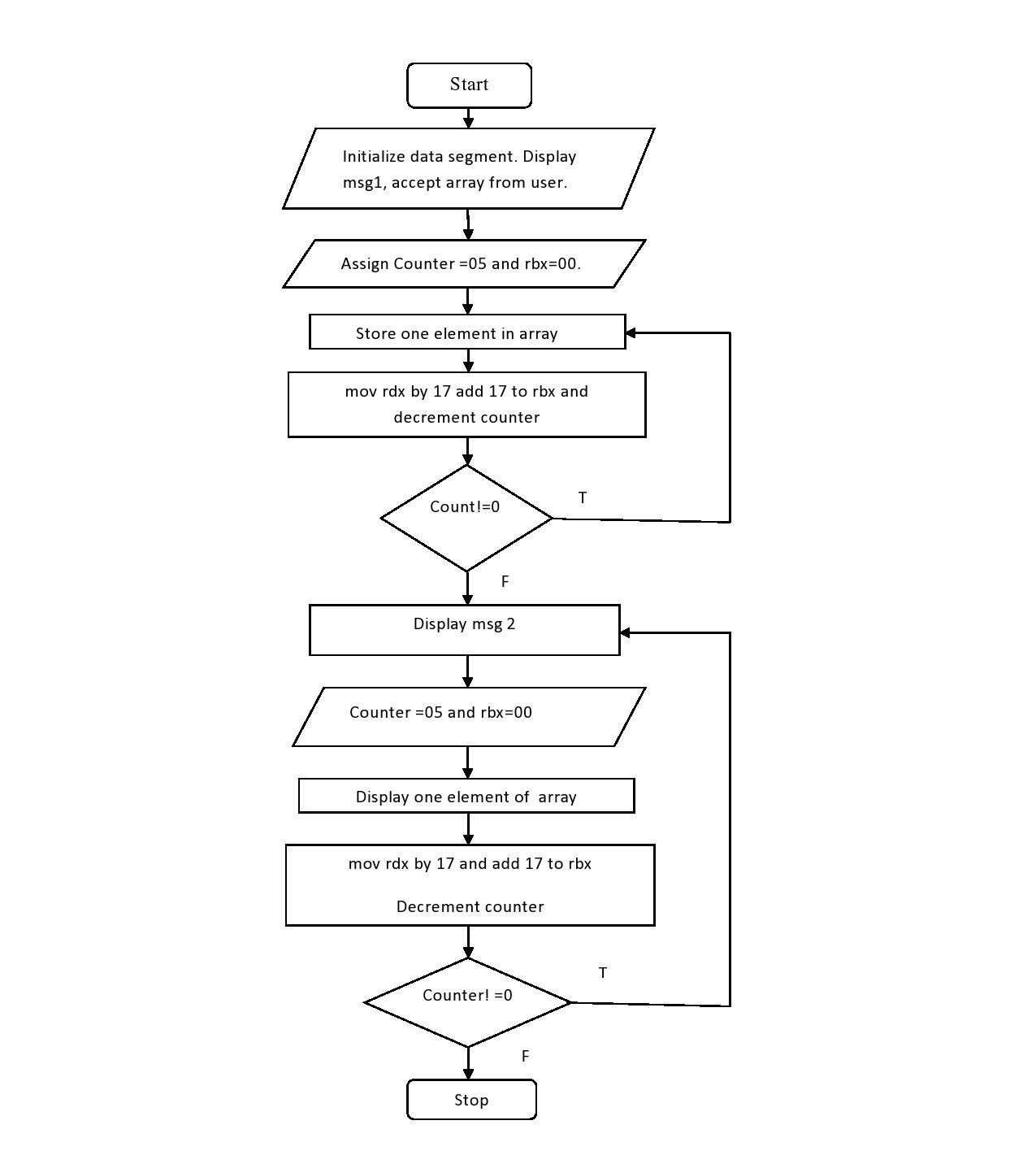
STEP 13: Add 17 to rbx.

STEP 14: Decrement Counter.

STEP 15: Jump to step 11 until counter value is not zero.

STEP 16: Stop

**FLOWCHART:**



**PROGRAM:**

section .data

msg1 db 10,13,"Enter 5 64 bit numbers" len1 equ $-msg1

msg2 db 10,13,"Entered 5 64 bit numbers"

len2 equ $-msg2

section .bss

array resd 200 counter resb 1 section .text

global \_start \_start:

;display

mov Rax,1 mov Rdi,1 mov Rsi,msg1 mov Rdx,len1

syscall

;accept mov byte[counter],05

mov rbx,00 loop1:

mov rax,0 ; 0 for read mov rdi,0 ; 0 for keyboard

mov rsi, array ;move pointer to start of array

add rsi,rbx

mov rdx,17

syscall

add rbx,17 ;to move counter

dec byte[counter]

JNZ loop1

;display

mov Rax,1 mov Rdi,1 mov Rsi,msg2 mov Rdx,len2

syscall

;display mov byte[counter],05

mov rbx,00 loop2:

mov rax,1 ;1 for write mov rdi, 1 ;1 for monitor mov rsi, array add rsi,rbx

mov rdx,17 ;16 bit +1 for enter syscall

add rbx,17

dec byte[counter]

JNZ loop2

;exit system call

mov rax ,60 mov rdi,0 syscall

;output

;vacoea@vacoea-Pegatron:~$ cd ~/Desktop

;vacoea@vacoea-Pegatron:~/Desktop$ nasm -f elf64 ass1.asm

;vacoea@vacoea-Pegatron:~/Desktop$ ld -o ass1 ass1.o

;vacoea@vacoea-Pegatron:~/Desktop$ ./ass1

;Enter 5 64 bit numbers12

;23

;34

;45

;56

;Entered 5 64 bit numbers12

;23

;34

;45

;56

**CONCLUSION:**

In this practical session we learnt how to write assembly language program and Accept and display array in assembly language.

**EXPERIMENT NO. 03**

**NAME:** Write an X86/64 ALP to accept a string and to display its length.

**AIM:** Write an X86/64 ALP to accept a string and to display its length.

**OBJECTIVES:**

* To understand assembly language programming instruction set.
* To understand different assembler directives with example.
* To apply instruction set for implementing X86/64 bit assembly language programs

**ENVIRONMENT:**

* Operating System: 64-bit Open source Linux or its derivative.
* Programming Tools: Preferably using Linux equivalent or MASM/TASM/NASM/FASM.  Text Editor: geditor

**THEORY:**

**MACRO:**

Writing a macro is another way of ensuring modular programming in assembly language.

* A macro is a sequence of instructions, assigned by a name and could be used anywhere in the program.
* In NASM, macros are defined with **%macro** and **%endmacro** directives.
* The macro begins with the %macro directive and ends with the %endmacro directive.

The Syntax for macro definition − %macro macro\_name number\_of\_params

<macro body>

%endmacro

Where, *number\_of\_params* specifies the number parameters, *macro\_name* specifies the name of the macro.

The macro is invoked by using the macro name along with the necessary parameters. When you need to use some sequence of instructions many times in a program, you can put those instructions in a macro and use it instead of writing the instructions all the time.

**PROCEDURE:**

Procedures or subroutines are very important in assembly language, as the assembly language programs tend to be large in size. Procedures are identified by a name. Following this name, the body of the procedure is described which performs a well-defined job. End of the procedure is indicated by a return statement.

Syntax

Following is the syntax to define a procedure −

|  |
| --- |
| proc\_name:  procedure body  ... ret |

The procedure is called from another function by using the CALL instruction. The CALL instruction should have the name of the called procedure as an argument as shown below − CALL proc\_name

The called procedure returns the control to the calling procedure by using the RET instruction.

**LIST OF INTERRRUPTS USED:** NA

**LIST OF ASSEMBLER DIRECTIVES USED**: EQU, PROC, GLOBAL, DB,

**LIST OF MACROS USED:** DISPMSG

**LIST OF PROCEDURES USED:** DISPLAY

**ALGORITHM:**

INPUT: String

OUTPUT: Length of String in hex STEP 1: Start.

STEP 2: Initialize data section.

STEP 3: Display msg1 on monitor

STEP 4: accept string from user and store it in Rsi Register (Its length gets stored in Rax register by default).

STEP 5: Display the result using “display” procedure. Load length of string in data register.

STEP 6. Take counter as 16 int cnt variable

STEP 7: move address of “result” variable into rdi.

STEP 8: Rotate left rbx register by 4 bit.

STEP 9: Move bl into al.

STEP 10: And al with 0fh

STEP 11: Compare al with 09h

STEP 12: If greater add 37h into al

STEP 13: else add 30h into al

STEP 14: Move al into memory location pointed by rdi

STEP 14: Increment rdi

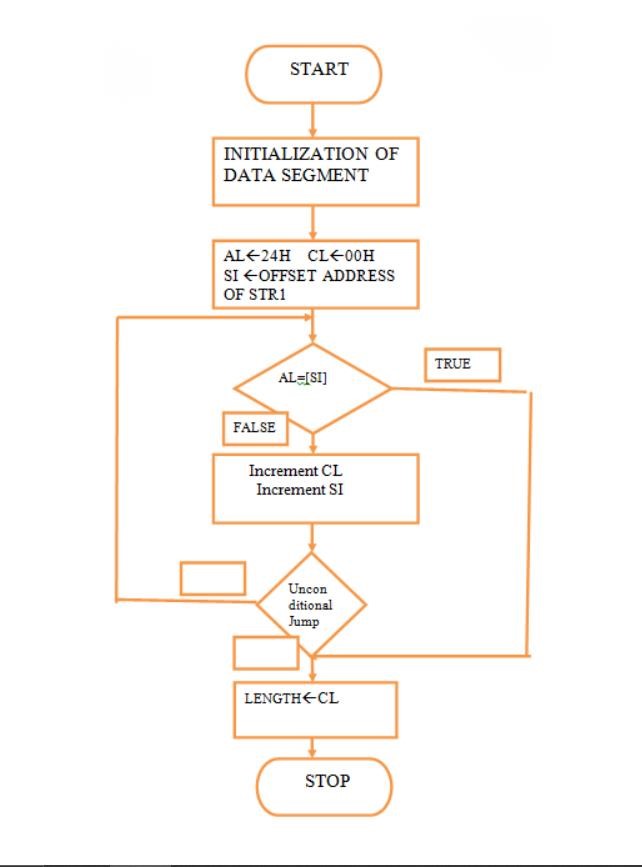
STEP 15: Loop the statement till counter value becomes zero

STEP 16: Call macro dispmsg and pass result variable and length to it. It will print length of string.

STEP 17: Return from procedure

STEP 18: Stop

**FLOWCHART:**



**PROGRAM:**

section .data

msg1 db 10,13,"Enter a string:"

len1 equ $-msg1

section .bss

str1 resb 200 ;string declaration result resb 16

section .text

global \_start

\_start:

;display

mov Rax,1 mov Rdi,1 mov Rsi,msg1 mov Rdx,len1 syscall

;store string

mov rax,0 mov rdi,0 mov rsi,str1 mov rdx,200

syscall

call display

;exit system call

mov Rax ,60 mov Rdi,0

syscall

%macro dispmsg 2 mov Rax,1 mov Rdi,1 mov rsi,%1 mov rdx,%2

syscall

%endmacro

display:

mov rbx,rax ; store no in rbx mov rdi,result ;point rdi to result variable mov cx,16 ;load count of rotation in cl up1:

rol rbx,04 ;rotate no of left by four bits

mov al,bl ; move lower byte in al

and al,0fh ;get only LSB cmp al,09h ;compare with 39h jg add\_37 ;if greater than 39h skip add 37

add al,30h

jmp skip ;else add 30 add\_37:

add al,37h skip:

mov [rdi],al ;store ascii code in result variable inc rdi ; point to next byte dec cx ; decrement counter jnz up1 ; if not zero jump to repeat

dispmsg result,16 ;call to macro ret

**CONCLUSION:**

In this practical session we learnt how to accept string and display its length.

**EXPERIMENT NO. 04**

**NAME:** Write an X86/64 ALP to count number of positive and negative numbers from the array.

**AIM:** Write an X86/64 ALP to count number of positive and negative numbers from the array.

**OBJECTIVES:**

* To understand assembly language programming instruction set.
* To understand different assembler directives with example.
* To apply instruction set for implementing X86/64 bit assembly language programs

**ENVIRONMENT:**

* Operating System: 64-bit Open source Linux or its derivative.
* Programming Tools: Preferably using Linux equivalent or MASM/TASM/NASM/FASM.  Text Editor: geditor

**THEORY:**

Mathematical numbers are generally made up of a sign and a value (magnitude) in which the sign indicates whether the number is positive, ( + ) or negative, ( – ) with the value indicating the size of the number, for example 23, +156 or -274. Presenting numbers is this fashion is called “sign-magnitude” representation since the left most digit can be used to indicate the sign and the remaining digits the magnitude or value of the number.

Sign-magnitude notation is the simplest and one of the most common methods of representing positive and negative numbers either side of zero, (0). Thus negative numbers are obtained simply by changing the sign of the corresponding positive number as each positive or unsigned number will have a signed opposite, for example, +2 and -2, +10 and -10, etc.

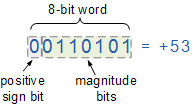
But how do we represent signed binary numbers if all we have is a bunch of one’s and zero’s. We know that binary digits, or bits only have two values, either a “1” or a “0” and conveniently for us, a sign also has only two values, being a “**+**” or a “**–**“.

Then we can use a single bit to identify the sign of a signed binary number as being positive or negative in value. So to represent a positive binary number (+n) and a negative (-n) binary number, we can use them with the addition of a sign.

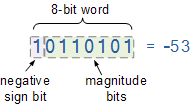
For signed binary numbers the most significant bit (MSB) is used as the sign bit. If the sign bit is “0”, this means the number is positive in value. If the sign bit is “1”, then the number is negative in value. The remaining bits in the number are used to represent the magnitude of the binary number in the usual unsigned binary number format way.

Then we can see that the Sign-and-Magnitude (SM) notation stores positive and negative values by dividing the “n” total bits into two parts: 1 bit for the sign and n–1 bits for the value which is a pure binary number. For example, the decimal number 53 can be expressed as an 8-bit signed binary number as follows.

**Positive Signed Binary Numbers**



**Negative Signed Binary Numbers**



**LIST OF INTERRRUPTS USED:** 80h

**LIST OF ASSEMBLER DIRECTIVES USED:** equ, db

**LIST OF MACROS USED:** print

**LIST OF PROCEDURES USED:** disp8num

**ALGORITHM:**

STEP 1: Initialize index register with the offset of array of signed numbers

STEP 2: Initialize ECX with array element count

STEP 3: Initialize positive number count and negative number count to zero

STEP 4: Perform MSB test of array element

STEP 5: If set jump to step 7

STEP 6: Else Increment positive number count and jump to step 8

STEP 7: Increment negative number count and continue

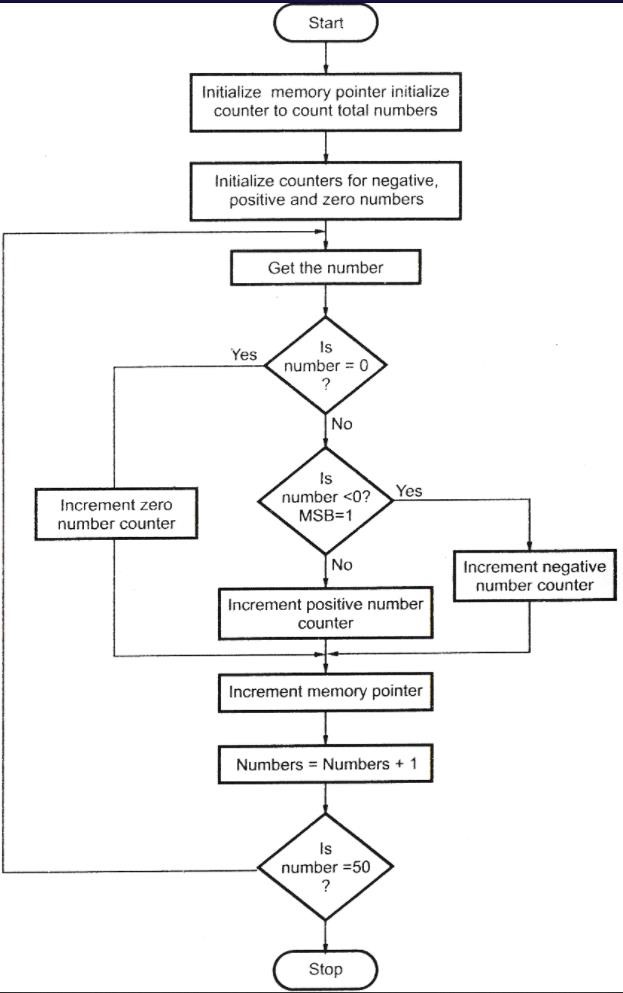
STEP 8: Point index register to the next element

STEP 9: Decrement the array element count from ECX, if not zero jump to step 4, else continue

STEP 10: Display Positive number message and then display positive number count

STEP 11: Display Negative number message and then display negative number count STEP 12: EXIT

**FLOWCHART:**



**PROGRAM:**

;Write an ALP to count no. of positive and negative numbers from the array.

section .data

welmsg db 10,'Welcome to count positive and negative numbers in an array',10 welmsg\_len equ $-welmsg

pmsg db 10,'Count of +ve numbers::'

pmsg\_len equ $-pmsg

nmsg db 10,'Count of -ve numbers::'

nmsg\_len equ $-nmsg

nwline db 10

array dw 8505h,90ffh,87h,88h,8a9fh,0adh,02h,8507h

arrcnt equ 8

pcnt db 0 ncnt db 0

section .bss

dispbuff resb 2

%macro print 2 ;defining print function

mov eax, 4 ; this 4 commands signifies the print sequence mov ebx, 1 mov ecx, %1 ; first parameter mov edx, %2 ;second parameter

int 80h ;interrupt command

%endmacro

section .text ;code segment

global \_start ;must be declared for linker

\_start: ;tells linker the entry point ;i.e start of code

print welmsg,welmsg\_len ;print title mov esi,array

mov ecx,arrcnt ;store array count in extended counter reg

up1: ;label bt word[esi],15

;bit test the array number (15th byte) pointed by esi.

;It sets the carray flag as the bit tested

jnc pnxt ;jump if no carry to label pskip

inc byte[ncnt] ;if the 15th bit is 1 it signifies it is a ;negative no and so we ;use this command to increment ncnt counter.

jmp pskip ;unconditional jump to label skip

pnxt: inc byte[pcnt] ;label pnxt if there no carry then it is ;positive no

;and so pcnt is incremented pskip: inc esi ;increment the source index but this ;instruction only increments it by 8 bit but the no’s in array ;are 16 bit word and hence it needs to be incremented twice.

inc esi

loop up1 ;loop it ends as soon as the array end “count” or

;ecx=0 loop automatically assums ecx has the counter

print pmsg,pmsg\_len ;prints pmsg

mov bl,[pcnt] ;move the positive no count to lower 8 bit of B reg call disp8num ;call disp8num subroutine print nmsg,nmsg\_len ;prints nmsg

mov bl,[ncnt] ;move the negative no count to lower 8 bits of b reg call disp8num ;call disp8num subroutine

print nwline,1 ;New line char

exit:

mov eax,01 mov ebx,0

int 80h

disp8num: mov ecx,2 ;move 2 in ecx ;Number digits to display mov edi,dispbuff ;Temp buffer

dup1: ;this command sequence which converts hex to bcd

rol bl,4 ;Rotate number from bl to get MS digit to LS digit mov al,bl ;Move bl i.e. rotated number to AL

and al,0fh ;Mask upper digit (logical AND the contents ;of lower8 bits of accumulator with 0fh )

cmp al,09 ;Compare al with 9

jbe dskip ;If number below or equal to 9 go to add only 30h ;add al,07h ;Else first add 07h to accumulator

dskip: add al,30h ;Add 30h to accumulator

mov [edi],al ;Store ASCII code in temp buff (move contents ;of accumulator to the location pointed by edi) inc edi

;Increment destination index i.e. pointer to ;next location in temp buff loop dup1 ;repeat till ecx becomes zero

print dispbuff,2 ;display the value from temp buff ret ;return to calling program

**OUTPUT:**

;[root@comppl2022 ~]# nasm -f elf64 Exp5.asm

;[root@comppl2022 ~]# ld -o Exp6 Exp5.o

;[root@comppl2022 ~]# ./Exp5

;Welcome to count +ve and -ve numbers in an array

;Count of +ve numbers::05

;Count of -ve numbers::03

;[root@comppl2022 ~]#

**CONCLUSION:**

In this practical session we learnt to count number of positive and negative numbers from the array.

**EXPERIMENT NO. 05**

**NAME:** Write an X86/64 ALP to find the largest of given Byte/Word/Dword/64-bit numbers

**AIM:** Write an X86/64 ALP to find the largest of given Byte/Word/Dword/64-bit numbers

**OBJECTIVES:**

* To understand assembly language programming instruction set.
* To understand different assembler directives with example.
* To apply instruction set for implementing X86/64 bit assembly language programs

**ENVIRONMENT:**

* Operating System: 64-bit Open source Linux or its derivative.
* Programming Tools: Preferably using Linux equivalent or MASM/TASM/NASM/FASM.  Text Editor: geditor

**THEORY:**

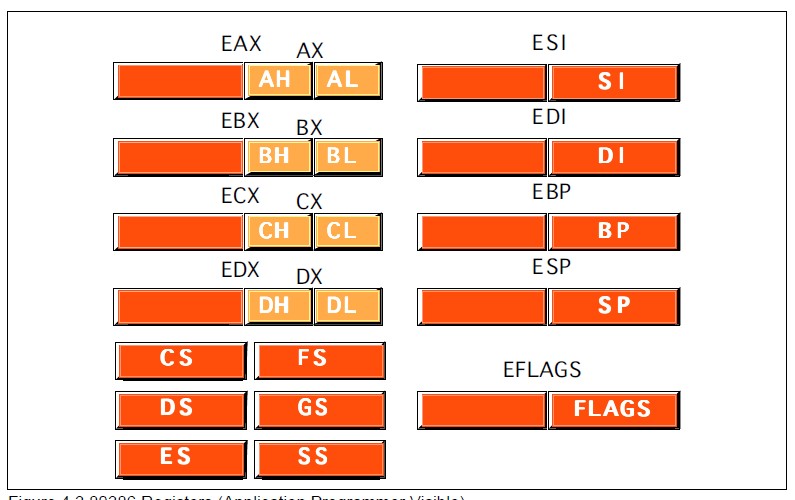
**Datatype in 80386:**

**Datatypes of 80386**:

The 80386 supports the following data types they are

* Bit
* Bit Field: A group of at the most 32 bits (4bytes)
* Bit String: A string of contiguous bits of maximum 4Gbytes in length.
* Signed Byte: Signed byte data  Unsigned Byte: Unsigned byte data.
* Integer word: Signed 16-bit data.
* Long Integer: 32-bit signed data represented in 2's complement form.
* Unsigned Integer Word: Unsigned 16-bit data
* Unsigned Long Integer: Unsigned 32-bit data
* Signed Quad Word: A signed 64-bit data or four word data.
* Unsigned Quad Word: An unsigned 64-bit data.
* Offset: 16/32-bit displacement that points a memory location using any of the addressing modes.
* Pointer: This consists of a pair of 16-bit selector and 16/32-bit offset.
* Character: An ASCII equivalent to any of the alphanumeric or control characters.
* Strings: These are the sequences of bytes, words or double words. A string may contain minimum one byte and maximum 4 Gigabytes.
* BCD: Decimal digits from 0-9 represented by unpacked bytes.
* Packed BCD: This represents two packed BCD digits using a byte, i.e. from 00 to 99.

**Registers in 80386:**



* General Purpose Register: EAX, EBX, ECX, EDX
* Pointer register: ESP, EBP
* Index register: ESI, EDI
* Segment Register: CS, FS, DS, GS, ES, SS
* Eflags register: EFLAGS
* System Address/Memory management Registers : GDTR, LDTR, IDTR
* Control Register: Cr0, Cr1, Cr2, Cr3
* Debug Register : DR0, DR,1 DR2, DR3, DR4, DR5, DR6, DR7  Test Register: TR0, TR,1 TR2, TR3, TR4, TR5, TR6, TR7

|  |  |  |
| --- | --- | --- |
| EAX | AX | AH,AL |
| EBX | BX | BH,BL |
| ECX | CX | CH,CL |
| EDX | DX | DH,DL |
| EBP | BP |  |
| EDI | DI |  |
| ESI | SI |  |
| ESP |  |  |

Size of operands in an Intel assembler instruction

* Specifying the size of an operand in Intel
* The size of the operand (byte, word, double word) is conveyed by the operand itself
* EAX means: a 32 bit operand
* AX means: a 16 bit operand
* AL means: a 8 bit operand The size of the source operand and the destination operand must be equal

**Addressing modes in 80386:**

The purpose of using addressing modes is as follows:

1. To give the programming versatility to the user.
2. To reduce the number of bits in addressing field of instruction.

|  |  |  |
| --- | --- | --- |
| 1. Register addressing mode: |  | MOV EAX, EDX |
| 2. Immediate Addressing modes: |  | MOV ECX, 20305060H |
| 3. Direct Addressing mode: |  | MOV AX, [1897 H] |
| 4. Register Indirect Addressing mode |  | MOV EBX, [ECX] |
| 5. Based Mode |  | MOV ESI, [EAX+23H] |
| 6. Index Mode |  | SUB COUNT [EDI], EAX |
| 7. Scaled Index Mode |  | MOV [ESI\*8], ECX |
| 8. Based Indexed Mode |  | MOV ESI, [ECX][EBX] |
| 9. Based Index Mode with displacement |  | EA=EBX+EBP+1245678H |

10. Based Scaled Index Mode with displacement MOV [EBX\*8] [ECX+5678H], ECX 11. String Addressing modes:

12. Implied Addressing modes:

**ALGORITHM:**

**Step 1:** Start.

**Step 2:** Initialize Block Size and get the address of first element.

**Step 3:** Load the data from the memory.

**Step 4**: Decrement Block Size and Increment address of first element.

**Step 5**: Store first element in A.

**Step 6**: Compare A with other elements, if A is smaller then store that element in B otherwise compare with next element.

**Step 7**: The value of B is the answer.

**Step 8**: Stop.

**PROGRAM:**

**32 Bit NASM Code for Byte**

%macro **scall** 4 ;macro declaration with 4 parameters

**mov** eax,%1 ;1st parameter has been moved to eax

**mov** ebx,%2 ;2nd parameter has been moved to ebx

**mov** ecx,%3 ;3rd parameter has been moved to ecx

**mov** edx,%4 ;4th parameter has been moved to edx **int** 80h ;Call the Kernel

%endmacro ;end of macro

section .data ;.data segment begins here m1 db "Enter size of array: ",10d,13d ;m1 initialised with a string

l1 equ $-m1 ;l1 stores length of m1 string

m2 db "Enter array elements: ",10d,13d ;m2 initialised with a string

l2 equ $-m2 ;l2 stores length of m2 string

m3 db 10d,13d,"Largest: " ;m3 initialised with a string

l3 equ $-m3 ;l3 stores length of m3 string

m4 db 10d,13d ;m4 initialised with a string

l4 equ $-m4 ;l4 stores length of m4 string section .bss ;.bss segment starts here cnt **resb** 3 ;variable with 3 byte size arr **resb** 3 ;variable with 3 byte size cnt1 **resb** 3 ;variable with 3 byte size arr1 **resb** 50 ;variable with 50 byte size temp **resb** 2 ;variable with 2 byte size char\_ans **resb** 2 ;variable with 2 byte size

section .text ;.text segment starts here global \_start ;declaring \_start label as global

\_start: ;\_start label

scall 4,1,m1,l1 ;macro call to display m1 scall 3,0,arr,3 ;macro call to input in arr

**mov** esi,arr ;esi points to arr **call** asciihextohex ;calling procedure to convert into hex nos

**mov** **byte**[cnt],dl ;moving values of dl into cnt

**mov** **byte**[cnt1],dl ;moving values of dl into cnt scall 4,1,m2,l2 ;macro call to display m2 **mov** edi,arr1 ;edi points to arr1

back: scall 3,0,arr,3 ;macro call to input in arr **mov** esi,arr ;esi points to arr **call** asciihextohex ;calling procedure to convert into hex nos

**mov** [edi],dl ;move contents of dl at address of edi

**inc** edi ;increment edi to point to next element

**dec** **byte**[cnt] ;decrement cnt variable **jnz** back ;jump if cnt is not zero to back label

**mov** esi,arr1 ;esi points to arr1 **mov** al,[esi] ;move contents at esi into al

**inc** esi ;increment esi to point to next element

up1: **mov** bl,[esi] ;move contents at esi into bl

**cmp** al,bl ;compare al with bl **jg** next1 ;if al is greater, jump to next1 **mov** **byte**[temp],al ;copying al into temp **mov** al,bl ;copying bl into al **mov** bl,**byte**[temp] ;copying temp into bl next1: **inc** esi ;increment esi **dec** **byte**[cnt1] ;decrement cnt1 **jnz** up1 ;jump to up1, if cnt1 not zero

**mov** ecx,02 ;copy 02 into ecx **mov** esi,char\_ans ;esi points to char\_ans

up4: **rol** al,4 ;roll contents of al by 4 bits

**mov** dl,al ;copy al into dl **and** dl,0FH ;AND dl with 0Fh **cmp** dl,09h ;compare dl with 09h **jbe** next2 ;jump to next2, if dl is below or equal

**add** dl,07h ;add 07h to dl next2: **add** dl,30h ;add 30h to dl **mov**[esi],dl ;move dl at esi address **inc** esi ;increment esi **dec** ecx ;decrement ecx **jnz** up4 ;jump to up4, if ecx not zero

scall 4,1,m3,l3 ;macro call to display m3 scall 4,1,char\_ans,2 ;macro call to display char\_ans

scall 4,1,m4,l4 ;macro call to display m4

**mov** eax,1 ;sys\_Exit

**mov** ebx,0 ;sucessfull termination **int** 80h ;call the kernel

asciihextohex: ;procedure **mov** ecx,2 ;copy 2 into ecx **mov** dl,0 ;copy 0 into dl

top: **rol** dl,4 ;roll contents of dl by 4 bits

**mov** al,[esi] ;copy esi contents into al **cmp** al,39h ;compare al with 39h **jbe** down ;jump to down, if al is below or equal

**sub** al,07h ;subtract 07h down: **sub** al,30h ;subtract 30h **add** dl,al ;add al with dl **inc** esi ;increment esi **loop** top ;jump if ecx not equal to zero,decrement ecx

**ret** ;return to calling address

**32 Bit NASM Code for Double Word**

%macro **scall** 4 ;macro declaration with 4 parameters

**mov** eax,%1 ;1st parameter has been moved to eax

**mov** ebx,%2 ;2nd parameter has been moved to ebx

**mov** ecx,%3 ;3rd parameter has been moved to ecx

**mov** edx,%4 ;4th parameter has been moved to edx **int** 80h ;Call the Kernel

%endmacro ;end of macro

section .data ;.data segment begins here m1 db "Enter size of array: ",10d,13d ;m1 initialised with a string

l1 equ $-m1 ;l1 stores length of m1 string

m2 db "Enter array elements: ",10d,13d ;m2 initialised with a string

l2 equ $-m2 ;l2 stores length of m2 string

m3 db 10d,13d,"Largest: " ;m3 initialised with a string

l3 equ $-m3 ;l3 stores length of m3 string

m4 db 10d,13d ;m4 initialised with a string

l4 equ $-m4 ;l4 stores length of m4 string

m5 db "Array Elements are: ",10d,13d l5 equ $-m5 section .bss

cnt **resb** 3 arr **resb** 9 cnt1 **resb** 3 cnt2 **resb** 2 arr1 **resb** 50 temp **resd** 1 char\_ans **resb** 8

section .text

global \_start

\_start:

;accept count scall 4,1,m1,l1 scall 3,0,arr,3 **mov** esi,arr **mov** ecx,2 **mov** dl,0 up: **rol** dl,4 **mov** al,[esi] **cmp** al,39h **jbe** L1 **sub** al,07h L1:

**sub** al,30h **add** dl,al **inc** esi **loop** up **mov** **byte**[cnt],dl **mov** **byte**[cnt1],dl **mov** **byte**[cnt2],dl

;accept array

scall 4,1,m2,l2 **mov** edi,arr1 back: scall 3,0,arr,9 **mov** esi,arr **mov** ecx,8 **mov** edx,0 **mov** eax,0 up1: **rol** edx,4 **mov** al,[esi] **cmp** al,39h **jbe** L2 **sub** al,07h L2:

**sub** al,30h **add** edx,eax **inc** esi **dec** ecx **jnz** up1 **mov** [edi],edx

**add** edi,9

**dec** **byte**[cnt] **jnz** back ;Displaying array elements scall 4,1,m5,l5 **mov** edi,arr1 up4:

**mov** ecx,08 **mov** esi,char\_ans **mov** eax,[edi] up6:

**rol** eax,4 **mov** dl,al **and** dl,0fh **cmp** dl,09h **jbe** nxt1 **add** dl,07h nxt1:

**add** dl,30h **mov** [esi],dl **inc** esi **dec** ecx **jnz** up6 scall 4,1,char\_ans,8 scall 4,1,m4,l4 **add** edi,9 **dec** **byte**[cnt2] **jnz** up4

;Finding Largest Number **mov** esi,arr1 **mov** eax,[esi] **add** esi,9 up2:

**mov** ebx,[esi] **cmp** eax,ebx **jg** next **mov** [temp],eax **mov** eax,ebx

**mov** ebx,[temp]

next: **add** esi,9 **dec** **byte**[cnt1] **jnz** up2

;Displaying Largest Number

**mov** ecx,08 **mov** esi,char\_ans up3:

**rol** eax,4

**mov** dl,al **and** dl,0fh **cmp** dl,09h

**jbe** nxt

**add** dl,07h nxt:

**dec** ecx **jnz** up3

**add**

dl,30h

**mov**

esi],dl

[

**inc**

esi

scall 4,1,m3,l3

scall 4,1,char\_ans,8

scall 4,1,m4,l4

**mov**

eax,1

**mov**

ebx,0

**int**

h

80

**64**

**Bit NASM Code for Quad Word**

%macro

**scall**

4

**mov**

1

rax,%

**mov**

rdi,%

2

**mov**

3

rsi,%

**mov**

4

rdx,%

**syscal**

**l**

%endmacr

o

section .data

m1 db "Enter Count of numbers: ",10

l1 equ $

-

m1

m2 db "Enter the numbers: ",10

l2 equ $

-

m2

m3 db "Largest Number is: ",10

l3 equ $

-

m3

m4 db " ",10

l4 equ $

-

m4

m5 db "Array Elements are: ",10

l5 equ $

-

m5

section .bss

cnt **resb** 3 arr **resb** 17 cnt1 **resb** 3 cnt2 **resb** 3 arr1 **resb** 100 temp **resq** 1

char\_ans **resb** 16

section .text global \_start

\_start:

;accept count scall 1,1,m1,l1 scall 0,0,arr,3 **mov** rsi,arr **mov** rcx,2 **mov** rdx,0 up: **rol** dl,4 **mov** al,[rsi] **cmp** al,39h **jbe** L1 **sub** al,07h L1:

**sub** al,30h **add** dl,al **inc** rsi **loop** up **mov** **byte**[cnt],dl **mov** **byte**[cnt1],dl **mov** **byte**[cnt2],dl

;accept array

scall 1,1,m2,l2 **mov** rbx,arr1 back: scall 0,0,arr,17 **mov** rsi,arr **mov** rcx,16 **mov** rdx,0 **mov** rax,0 up1: **rol** rdx,4 **mov** al,[rsi] **cmp** al,39h **jbe** L2 **sub** al,07h

L2: **sub** al,30h

**add** rdx,rax

**inc** rsi

**dec** rcx **jnz** up1

**mov** [rbx],rdx

**add** rbx,17 **dec** **byte**[cnt]

**jnz** back

;\*\*\*\*\*\*\*\*\*\*\*Displaying array elements\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* scall 1,1,m5,l5 **mov** rbx,arr1 up4:

**mov** rcx,16 **mov** rsi,char\_ans **mov** rax,[rbx] up6: **rol** rax,4 **mov** dl,al **and** dl,0Fh

**cmp** dl,09h **jbe** nxt1 **add** dl,07h nxt1: **add** dl,30h

**mov** [rsi],dl

**inc** rsi **dec** rcx **jnz** up6

scall 1,1,char\_ans,16

scall 1,1,m4,l4

**add** rbx,17 **dec** **byte**[cnt2] **jnz** up4

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Finding Largest Number\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* **mov** rsi,arr1 **mov** rax,[rsi] **add** rsi,17 up2:

**mov** rbx,[rsi] **cmp** rax,rbx **jg** next **mov** [temp],rax **mov** rax,rbx

**mov** rbx,[temp]

next:

**add** rsi,17 **dec** **byte**[cnt1] **jnz** up2

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Displaying Largest

Number\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**mov** rcx,16 **mov** rsi,char\_ans up3:

**rol** rax,4 **mov** dl,al **and** dl,0fh **cmp** dl,09h

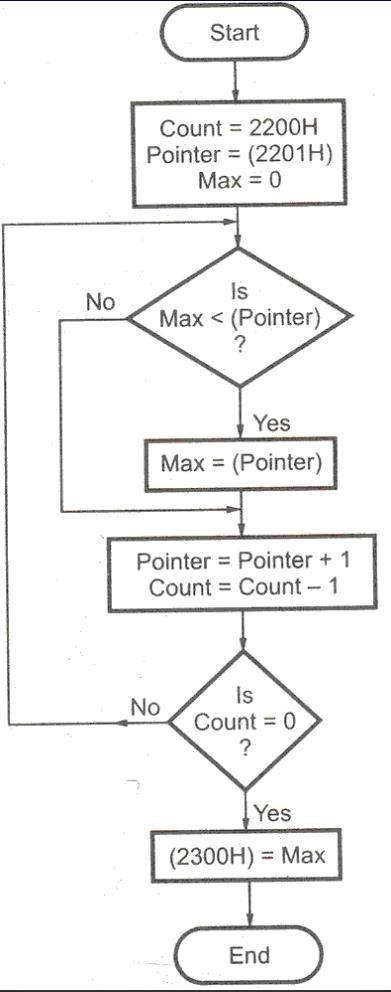
**jbe** nxt

**add** dl,07h nxt: **add** dl,30h **mov** [rsi],dl **inc** rsi **dec** rcx **jnz** up3 scall 1,1,m3,l3 scall 1,1,char\_ans,16 scall 1,1,m4,l4

**mov** rax,60 **mov** rbx,0

**syscall**

**FLOWCHART:**



**CONCLUSION:** In this practical session we learnt to find the largest of given Byte / Word / Dword / 64-Bit Numbers.

**EXPERIMENT NO. 06**

**NAME:** Write X86/64 ALP to detect protected mode and display the values of GDTR, LDTR, IDTR, TR and MSW Registers also identify CPU type using CPUID instruction.

**AIM:** Write X86/64 ALP to detect protected mode and display the values of GDTR, LDTR, IDTR, TR and MSW Registers also identify CPU type using CPUID instruction.

**OBJECTIVES:**

* To understand assembly language programming instruction set.
* To understand different assembler directives with example.
* To apply instruction set for implementing X86/64 bit assembly language programs

**ENVIRONMENT:**

* Operating System: 64-bit Open source Linux or its derivative.
* Programming Tools: Preferably using Linux equivalent or MASM/TASM/NASM/FASM.  Text Editor: geditor

**THEORY:**

**Real Mode:**

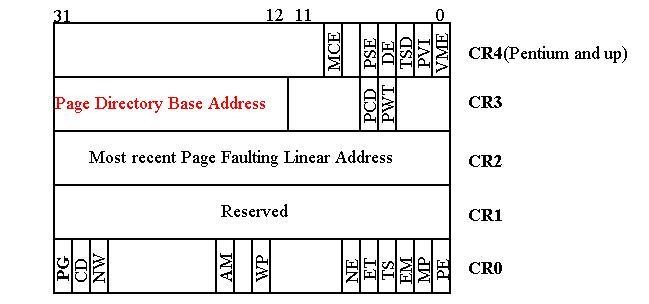
Real mode, also called real address mode, is an operating mode of all x86-compatible CPUs. Real mode is characterized by a 20-bit segmented memory address space (giving exactly 1 MiB of addressable memory) and unlimited direct software access to all addressable memory, I/O addresses and peripheral hardware. Real mode provides no support for memory protection, multitasking, or code privilege levels.

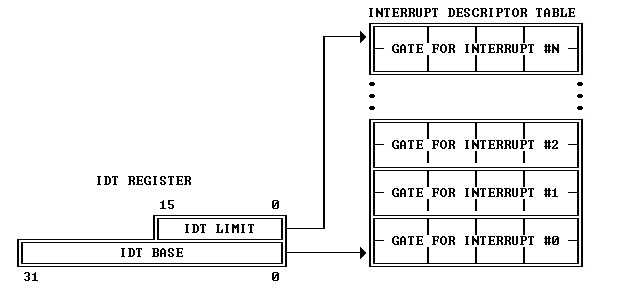
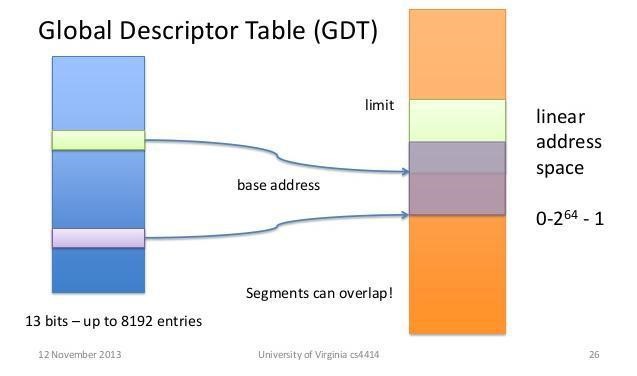
**Protected Mode:**

In computing, protected mode, also called protected virtual address mode is an operational mode of x86-compatible central processing units (CPUs). It allows system software to use features such as virtual memory, paging and safe multi-tasking designed to increase an operating system's control over application software.

When a processor that supports x86 protected mode is powered on, it begins executing instructions in real mode, in order to maintain backward compatibility with earlier x86 processors. Protected mode may only be entered after the system software sets up several descriptor tables and enables the Protection Enable (PE) bit in the control register 0 (CR0).

**Control Register :**





**Global Descriptor Table Register**

This register holds the 32

-

bit base address and 16

-

bit segment limit for the global descriptor table

(

GDT). When a reference is made to data in memory, a segment selector is used to find a segment

des

criptor in the GDT or LDT. A segment descriptor contains the base address for a segment.

**Local Descriptor Table Register**

This register holds the 32-bit base address, 16-bit segment limit, and 16-bit segment selector for the local descriptor table (LDT). The segment which contains the LDT has a segment descriptor in the GDT. There is no segment descriptor for the GDT. When a reference is made to data in memory, a segment selector is used to find a segment descriptor in the GDT or LDT. A segment descriptor contains the base address for a segment

**Interrupt Descriptor Table Register**

This register holds the 32-bit base address and 16-bit segment limit for the interrupt descriptor table (IDT). When an interrupt occurs, the interrupt vector is used as an index to get a gate descriptor from this table. The gate descriptor contains a far pointer used to start up the interrupt handler.

**ALGORITHM:**

* Start
* Display the message using sys\_write call
* Read CR0
* Checking PE bit, if 1=Protected Mode
* Load number of digits to display
* Rotate number left by four bits
* Convert the number in ASCII
* Display the number from buffer
* Exit using sys\_exit call

**PROGRAM:**

section .data

rmodemsg db 10,'Processor is in Real Mode'

rmsg\_len:equ $-rmodemsg

pmodemsg db 10,'Processor is in Protected Mode' pmsg\_len:equ $-pmodemsg

gdtmsg db 10,'GDT Contents are::' gmsg\_len:equ $-gdtmsg

ldtmsg db 10,'LDT Contents are::' lmsg\_len:equ $-ldtmsg

idtmsg db 10,'IDT Contents are::'

imsg\_len:equ $-idtmsg

trmsg db 10,'Task Register Contents are::' tmsg\_len: equ $-trmsg

mswmsg db 10,'Machine Status Word::'

mmsg\_len:equ $-mswmsg

colmsg db ':'

nwline db 10

section .bss gdt resd 1 resw 1 ldt resw 1 idt resd 1 resw 1 tr resw 1

cr0\_data resd 1

dnum\_buff resb 04

%macro print 2 mov rax,01 mov rdi,01 mov rsi,%1

mov rdx,%2

syscall

%endmacro

section .text global \_start \_start: smsw eax ;Reading CR0. As MSW is 32-bit cannot use RAX register.

mov [cr0\_data],rax

bt rax,1 ;Checking PE bit, if 1=Protected Mode, else Real Mode jc prmode print rmodemsg,rmsg\_len jmp nxt1

prmode: print pmodemsg,pmsg\_len

nxt1: sgdt [gdt] sldt [ldt] sidt [idt] str [tr]

print gdtmsg,gmsg\_len

mov bx,[gdt+4]

call print\_num

mov bx,[gdt+2]

call print\_num

print colmsg,1

mov bx,[gdt]

call print\_num

print ldtmsg,lmsg\_len mov bx,[ldt] call print\_num

print idtmsg,imsg\_len

mov bx,[idt+4]

call print\_num

mov bx,[idt+2] call print\_num

print colmsg,1

mov bx,[idt]

call print\_num

print trmsg,tmsg\_len

mov bx,[tr]

call print\_num

print mswmsg,mmsg\_len

mov bx,[cr0\_data+2]

call print\_num

mov bx,[cr0\_data]

call print\_num

print nwline,1

exit: mov rax,60 xor rdi,rdi

syscall

print\_num: mov rsi,dnum\_buff ;point esi to buffer

mov rcx,04 ;load number of digits to printlay

up1: rol bx,4 ;rotate number left by four bits mov dl,bl ;move lower byte in dl and dl,0fh ;mask upper digit of byte in dl add dl,30h ;add 30h to calculate ASCII code cmp dl,39h ;compare with 39h jbe skip1 ;if less than 39h skip adding 07 more add dl,07h ;else add 07 skip1:

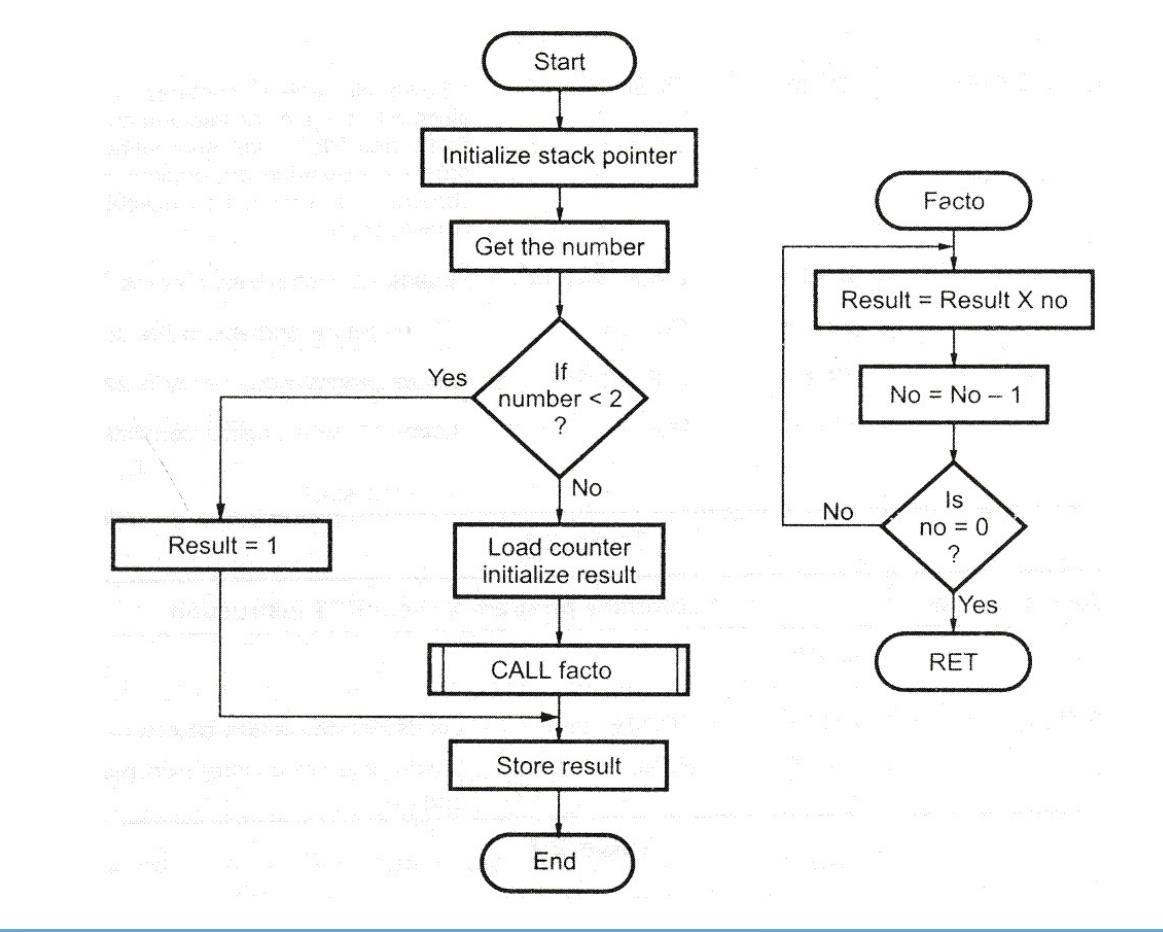
mov [rsi],dl ;store ASCII code in buffer inc rsi ;point to next byte

loop up1 ;decrement the count of digits to printlay

;if not zero jump to repeat

print dnum\_buff,4 ;printlay the number from buffer ret

**FLOWCHART:**



**CONCLUSION:**

In this practical session we learnt to detect protected mode and display the values of GDTR, LDTR, IDTR, TR and MSW Registers also identified CPU type using CPUID instruction.

# EXPERIMENT NO. 07

**NAME:** Write X86/64 ALP to perform non-overlapped block transfer without string specific instructions. Block containing data can be defined in the data segment.

**AIM:** Write X86/64 ALP to perform non-overlapped block transfer without string specific instructions. Block containing data can be defined in the data segment.

**OBJECTIVES:**

* To understand assembly language programming instruction set.
* To understand different assembler directives with example.
* To apply instruction set for implementing X86/64 bit assembly language programs

**ENVIRONMENT:**

* Operating System: 64-bit Open source Linux or its derivative.
* Programming Tools: Preferably using Linux equivalent or

MASM/TASM/NASM/FASM.

* Text Editor: geditor

**THEORY:**

* Consider that a block of data of N bytes is present at source location. Now this block of N bytes is to be moved from source location to a destination location.



* Let the number of bytes N = 05.



* We will have to initialize this as count.



* We know that source address is in the ESI register and destination address is in the EDI

register.



* For block transfer without string instruction, move contents at ESI to accumulator and from accumulator to memory location of EDI and increment ESI and EDI for next content transfer.



* For block transfer with string instruction, clear the direction flag. Move the data from source location to the destination location using string instruction.

**Instructions needed:**

1. **MOVSB**-This is a string instruction and it moves string byte from source to destination.

1. **REP-** This is prefix that are applied to string operation. Each prefix cause the stringinstruction that follows to be repeated the number of times indicated in the count register.
2. **CLD-** Clear Direction flag. ESI and EDI will be incremented and DF = 0

1. **STD-** Set Direction flag. ESI and EDI will be incremented and DF = 1

1. **ROL**-Rotates bits of byte or word left.

1. **AND**-AND each bit in a byte or word with corresponding bit in another byte or word.

1. **INC**-Increments specified byte/word by1.

1. **DEC**-Decrements specified byte/word by1.

1. **JNZ**-Jumps if not equal to Zero.

1. **JNC**-Jumps if no carry is generated.

1. **CMP**-Compares to specified bytes or words.

1. **JBE**-Jumps if below or equal.

1. **ADD**-Adds specified byte to byte or word to word.

1. **CALL**-Transfers the control from calling program to procedure.

1. **RET**-Return from where call is made.

**ALGORITHM:**

1. Initialize ESI and EDI with source and destination address.

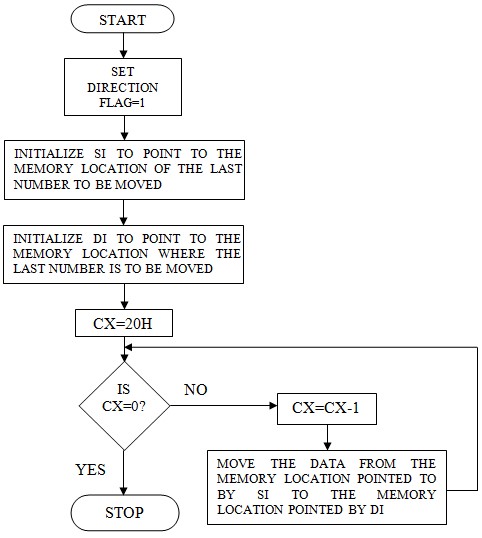
1. Move count in ECX register.

1. Move contents at ESI to accumulator and from accumulator to memory location of EDI.

1. Increment ESI and EDI to transfer next content.

1. Repeat procedure till count becomes zero.

**FLOWCHART:**



**PROGRAM:**

section .data

array db 10h,20h,30h,40h,50h

msg1: db 'Before overlapped :',0xa len1: equ $-msg1

msg2: db 'After overlapped :',0xa len2: equ $-msg2

msg3: db ' ',0xa len3: equ $-msg3

msg4: db ' : ' len4: equ $-msg4

count db 0 count1 db 0 count2 db 0 count3 db 0

count4 db 0

section .bss addr resb 16

num1 resb 2

section .text global \_start

\_start:

mov rax,1 mov rdi,1 mov rsi,msg1 mov rdx,len1

syscall

xor rsi,rsi

mov rsi,array mov byte[count],05

up: mov rbx,rsi push rsi mov rdi,addr call HtoA1 pop rsi

mov dl,[rsi] push rsi mov rdi,num1

call HtoA2 pop rsi

inc rsi

dec byte[count]

jnz up

mov rsi,array mov rdi,array+5h mov byte[count3],05h

loop10: mov dl,00h mov dl,byte[rsi] mov byte[rdi],dl inc rsi inc rdi dec byte[count3]

jnz loop10

mov rax,1 mov rdi,1 mov rsi,msg2 mov rdx,len2

syscall

mov rsi,array

mov byte[count4],0Ah

up10: mov rbx,rsi push rsi mov rdi,addr call HtoA1

pop rsi

mov dl,[rsi] push rsi mov rdi,num1 call HtoA2

pop rsi

inc rsi

dec byte[count4] jnz up10

mov rax,60 mov rdi,0 syscall

HtoA1: mov byte[count1],16

dup1: rol rbx,4 mov al,bl and al,0fh cmp al,09 jg p3 add al,30h jmp p4 p3: add al,37h p4:mov [rdi],al inc rdi dec byte[count1] jnz dup1

mov rax,1 mov rdi,1 mov rsi,addr mov rdx,16

syscall

mov rax,1 mov rdi,1 mov rsi,msg4 mov rdx,len4 syscall

ret

HtoA2: mov byte[count2],02

dup2:

rol dl,04 mov al,dl and al,0fh cmp al,09h jg p31 add al,30h

jmp p41

p31: add al,37h p41:mov [rdi],al

inc rdi dec byte[count2] jnz dup2

mov rax,1 mov rdi,1 mov rsi,num1 mov rdx,02

syscall

mov rax,1 mov rdi,1 mov rsi,msg3 mov rdx,len3 syscall

ret

OUTPUT:

; nasm -f elf64 nonover\_string.asm

;ld -o nonover\_string nonover\_string.o

;./nonover\_string

;Before overlapped :

;0000000000600270 : 10 ;0000000000600271 : 20

;0000000000600272 : 30

;0000000000600273 : 40

;0000000000600274 : 50

;After overlapped :

;0000000000600270 : 10

;0000000000600271 : 20

;0000000000600272 : 30

;0000000000600273 : 40

;0000000000600274 : 50

;0000000000600275 : 10

;0000000000600276 : 20

;0000000000600277 : 30

;0000000000600278 : 40

;0000000000600279 : 50

**CONCLUSION:**

In this practical session we learnt how to perform non-overlapped block transfer without string specific

instructions.

**EXPERIMENT NO. 08**

**NAME:** Write X86/64 ALP to perform overlapped block transfer with string specific instructions

Block containing data can be defined in the data segment.

**AIM:** Write X86/64 ALP to perform overlapped block transfer with string specific instructions Block containing data can be defined in the data segment.

**OBJECTIVES:**

* To understand assembly language programming instruction set.
* To understand different assembler directives with example.
* To apply instruction set for implementing X86/64 bit assembly language programs

**ENVIRONMENT:**

* Operating System: 64-bit Open source Linux or its derivative.
* Programming Tools: Preferably using Linux equivalent or

MASM/TASM/NASM/FASM.

* Text Editor: geditor

**THEORY:**

* Consider that a block of data of N bytes is present at source location. Now this block of N bytes is to be moved from source location to a destination location.



* Let the number of bytes N = 05.



* We will have to initialize this as count.



* Overlap the source block and destination block.



* We know that source address is in the ESI register and destination address is in the EDI

register.



* For block transfer without string instruction, move contents at ESI to accumulator and from accumulator to memory location of EDI and decrement ESI and EDI for next content transfer.



* For block transfer with string instruction, set the direction flag. Move the data from source location to the destination location using string instruction.

**Instructions needed:**

1. **MOVSB**-This is a string instruction and it moves string byte from source to destination.

1. **REP-** This is prefix that are applied to string operation. Each prefix cause the stringinstruction that follows to be repeated the number of times indicated in the count register.
2. **CLD-** Clear Direction flag. ESI and EDI will be incremented and DF = 0

1. **STD-** Set Direction flag. ESI and EDI will be decremented and DF = 1

1. **ROL**-Rotates bits of byte or word left.

1. **AND**-AND each bit in a byte or word with corresponding bit in another byte or word.

1. **INC**-Increments specified byte/word by1.

1. **DEC**-Decrements specified byte/word by1.

1. **JNZ**-Jumps if not equal to Zero.

1. **JNC**-Jumps if no carry is generated.

1. **CMP**-Compares to specified bytes or words.

1. **JBE**-Jumps if below or equal.

1. **ADD**-Adds specified byte to byte or word to word.

1. **CALL**-Transfers the control from calling program to procedure.

1. **RET**-Return from where call is made.

**ALGORITHM:**

1. Initialize ESI and EDI with source and destination address.

1. Move count in ECX register.

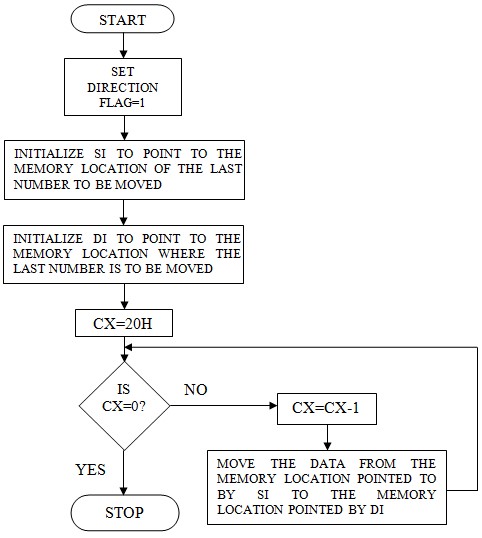
1. Move source block’s and destination block’s last content address in ESI and EDI.

1. Move contents at ESI to accumulator and from accumulator to memory location of EDI.

1. Decrement ESI and EDI to transfer next content.

1. Repeat procedure till count becomes zero.

**FLOWCHART:**



**PROGRAM:**

section .data

array db 10h,20h,30h,40h,50h

msg1: db 'Before overlapped :',0xa

len1: equ $-msg1

msg2: db 'After overlapped :',0xa

len2: equ $-msg2

msg3: db ' ',0xa

len3: equ $-msg3

msg4: db ' : ' len4: equ $-msg4

count db 0 count1 db 0 count2 db 0 count3 db 0 count4 db 0 count5 db 0

section .bss addr resb 16

num1 resb 2

section .text global \_start

\_start:

mov rax,1 mov rdi,1 mov rsi,msg1

mov rdx,len1 syscall

xor rsi,rsi

mov rsi,array mov byte[count],05

up: mov rbx,rsi push rsi mov rdi,addr call HtoA1 pop rsi

mov dl,[rsi] push rsi mov rdi,num1 call HtoA2 pop rsi

inc rsi

dec byte[count] jnz up

mov rsi,array mov rdi,array+0Ah

mov byte[count3],05h

loop10: mov dl,00h mov dl,byte[rsi] mov byte[rdi],dl inc rsi inc rdi dec byte[count3]

jnz loop10

xor rsi,rsi

mov rsi,array+3h mov rdi,array+0Ah mov byte[count5],05h

loop11:

mov dl,byte[rdi] mov byte[rsi],dl inc rsi inc rdi dec byte[count5]

jnz loop11

mov rax,1 mov rdi,1 mov rsi,msg2 mov rdx,len2

syscall

xor rsi,rsi mov rsi,array mov byte[count4],08h

up10: mov rbx,rsi push rsi mov rdi,addr call HtoA1 pop rsi

mov dl,[rsi] push rsi mov rdi,num1 call HtoA2 pop rsi

inc rsi

dec byte[count4] jnz up10

mov rax,60 mov rdi,0 syscall

HtoA1: mov byte[count1],16 dup1: rol rbx,4 mov al,bl and al,0fh cmp al,09 jg p3 add al,30h

jmp p4 p3: add al,37h p4:mov [rdi],al inc rdi dec byte[count1] jnz dup1

mov rax,1 mov rdi,1 mov rsi,addr mov rdx,16 syscall

mov rax,1 mov rdi,1 mov rsi,msg4 mov rdx,len4 syscall

ret

HtoA2: mov byte[count2],02

dup2:

rol dl,04 mov al,dl and al,0fh cmp al,09h jg p31 add al,30h

jmp p41

p31: add al,37h p41:mov [rdi],al inc rdi dec byte[count2]

jnz dup2

mov rax,1 mov rdi,1

mov rsi,num1 mov rdx,02

syscall

mov rax,1 mov rdi,1 mov rsi,msg3 mov rdx,len3

syscall

ret

OUTPUT:

;nasm -f elf64 with\_over.asm

; ld -o with\_over with\_over.o

;./with\_over

;Before overlapped :

;00000000006002A4 : 10

;00000000006002A5 : 20

;00000000006002A6 : 30

;00000000006002A7 : 40

;00000000006002A8 : 50 ;After overlapped :

;00000000006002A4 : 10

;00000000006002A5 : 20

;00000000006002A6 : 30

;00000000006002A7 : 10

;00000000006002A8 : 20

;00000000006002A9 : 30

;00000000006002AA : 40

;00000000006002AB : 50

**CONCLUSION:** In this practical session we learnt how to perform non-overlapped block transfer with string specific instructions.

**EXPERIMENT NO. 09**

**NAME:** Write X86/64 ALP to perform multiplication of two 8-bit hexadecimal numbers. Use

successive addition and add and shift method. (use of 64-bit registers is expected).

**AIM:** Write X86/64 ALP to perform multiplication of two 8-bit hexadecimal numbers. Use

successive addition and add and shift method. (use of 64-bit registers is expected).

**OBJECTIVES:**

* To understand assembly language programming instruction set.
* To understand different assembler directives with example.
* To apply instruction set for implementing X86/64 bit assembly language programs

**ENVIRONMENT:**

* Operating System: 64-bit Open source Linux or its derivative.
* Programming Tools: Preferably using Linux equivalent or

MASM/TASM/NASM/FASM.

* Text Editor: geditor

**THEORY:**

**A) Multiplication of two numbers using successive addition method:**

Historically, computers used a "shift and add" algorithm for multiplying small integers. Both base 2 long multiplication and base 2 peasant multiplications reduce to this same algorithm. In base 2, multiplying by the single digit of the multiplier reduces to a simple series of logical AND operations. Each partial product is added to a running sum as soon as each partial product is computed. Most currently available microprocessors implement this or other similar algorithms (such as Booth encoding) for various integer and floating-point sizes in hardware multipliers or in microcode.

On currently available processors, a bit-wise shift instruction is faster than a multiply instruction and can be used to multiply (shift left) and divide (shift right) by powers of two. Multiplication by a constant and division by a constant can be implemented using a sequence of shifts and adds or subtracts. For example, there are several ways to multiply by 10 using only bit-shift and addition.

((x << 2) + x) << 1 # Here 10\*x is computed as (x\*2^2 + x)\*2

(X << 3) + (x << 1) # Here 10\*x is computed as x\*2^3 + x\*2

In some cases such sequences of shifts and adds or subtracts will outperform hardware multipliers and especially dividers. A division by a number of the form 2n or 2n ±1 often can be converted to such a short sequence. These types of sequences have to always be used for computers that do not have a "multiply" instruction,[4] and can also be used by extension to floating point numbers if one replaces the shifts with computation of *2\*x* as *x+x*, as these are logically equivalent.

**Example:**

Consider that a byte is present in the AL register and second byte is present in the BL register.

**Step 1:** We have to multiply the byte in AL with the byte in BL.

**Step 2:** We will multiply the numbers using successive addition method.

**Step 3:** In successive addition method, one number is accepted and othernumber is taken as a counter. The first number is added with itself, till the counter decrements to zero.

**Step 4:** Result is stored in DX register. Display the result, using display routine.

**For example:** AL = 12 H, BL = 10 H **Solution:**

Result = 12H + 12H + 12H + 12H + 12H + 12H + 12H + 12H + 12H+ 12H Result = 0120 H

**Algorithm to Multiply Two 8 Bit Numbers Successive Addition Method:**

**Step I** **:** Initialize the data segment.

**Step II** **:** Get the first number.

**Step III** **:** Get the second number as counter.

**Step IV :** Initialize result = 0.

**Step V :** Result = Result + First number.

**Step VI :** Decrement counter

**Step VII :** If count ¹ 0, go to step V.

**Step VIII :** Display the result.

**Step IX :** Stop.

**B) Multiply Two 8 Bit Numbers using Add and Shift Method:**

Program should take first and second numbers as input to the program. Now it should implement certain logic to multiply 8 bit Numbers using Add and Shift Method. Consider that one byte is present in the AL register and another byte is present in the BL register. We have to multiply the byte in AL with the byte in BL.

**Steps for multiply the numbers using add and shift method:**

**Step 1:** In this method, you add number with itself

**Step 2:** Rotate the other number each time and shift it by one bit to leftalong with carry. If carry is present add the two numbers.

**Step 3:** Initialize the count to 4 as we are scanning for 4 digits. Decrementcounter each time the bits are added. The result is stored in AX. Display the result.

**For example:** AL = 11 H, BL = 10 H, Count = 4  **Solution:**

**Step I** **: AX= 11**

+ **11**

22H

Rotate BL by one bit to left along with carry

BL=10 H 0 0001 0000

CY 10

After Rotate BL by one bit to left along with carry

BL= 0 0010 0000

CY 20

**Step II** **:** Now decrement counter count = 3.

Check for carry, carry is not there so add number with itself.

**AX=22**

+ **22**

44H

Rotate BL to left,

BL= 0 0010 0000

CY 20

After Rotate BL by one bit to left along with carry

BL= 0 0100 0000

CY 40

Carry is not there.

Decrement count, count=2

**Step III :** Add number with itself

**AX=44**

+ **44**

88H

Rotate BL to left,

BL= 0 0100 0000

CY 40

After Rotate BL by one bit to left along with carry

BL= 0 1000 0000

CY 80

Carry is not there.

**Step IV :** Decrement counter count = 1.

Add number with itself as carry is not there.

**AX=88**

+ **88**

110H

Rotate BL to left,

BL= 0 1000 0000

CY 80

After Rotate BL by one bit to left along with carry

BL= 1 0000 0000

CY 00

Carry is there.

**Step V :** Decrement counter = 0.

Carry is present.

\ add AX, BX

|  |  |
| --- | --- |
| 0110 | i.e.11 H |
| + 0000 | i.e.10 H |
| 0110 H | 0110H |

**Algorithm to Multiply Two 8 Bit Numbers using Add and Shift Method:**

**Step I :** Get the first number.

**Step II :** Get the second number

|  |  |
| --- | --- |
| **Step III** | **:** Initialize count = 04. |
| **Step IV** | **:** number 1 = number 1 ´ 2. |
| **Step V** | **:** Shift multiplier to left along with carry. |
| **Step VI** | **:** Check for carry, if present go to step VIII else go to step IX. |

**Step VII :** number 1 = number1 + shifted number 2.

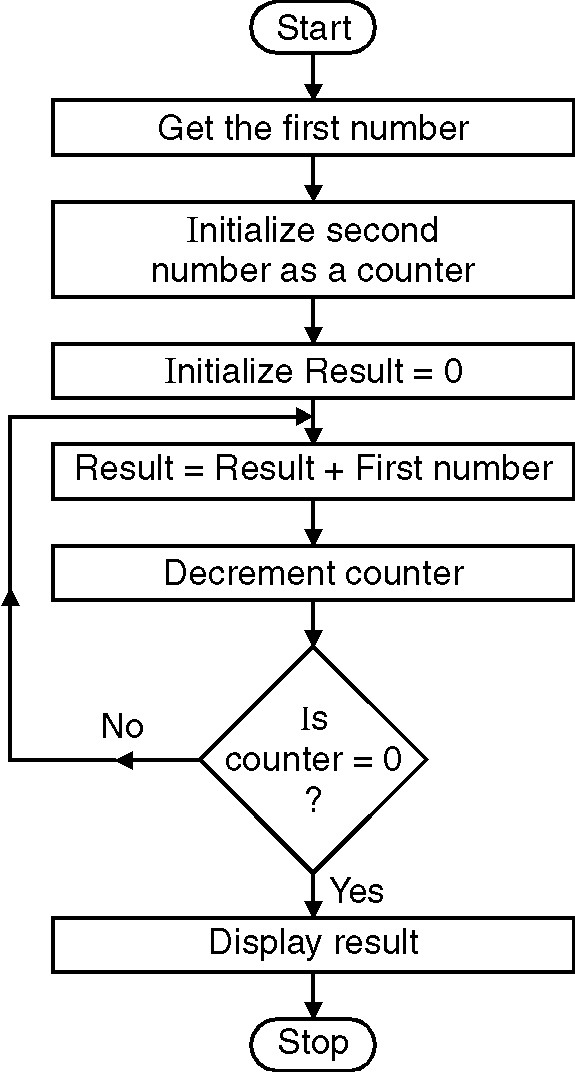
**Step VIII :** Decrement counter.

**Step IX :** If not zero, go to step V.

**Step X :** Display the result.

**Step XI :** Stop.

**FLOWCHART:**



**PROGRAM:**

section .data

msg db 'Enter two digit Number::',0xa msg\_len equ $-msg

res db 10,'Multiplication of elements is::' res\_len equ $-res choice db 'Enter your Choice:',0xa db'1.Successive Addition',0xa db '2.Add and Shift method',0xa

db '3.Exit',0xa

choice\_len equ $-choice

section .bss num resb 03 num1 resb 01 result resb 04 cho resb 2

section .text

global \_start \_start:

xor rax,rax xor rbx,rbx xor rcx,rcx xor rdx,rdx mov byte[result],0 mov byte[num],0 mov byte[num1],0

mov rax,1 mov rdi,1 mov rsi,choice mov rdx,choice\_len

syscall

mov rax,0 ;; read choice mov rdi,0

mov rsi,cho mov rdx,2

syscall

cmp byte[cho],31h ;; comparing choice je a

cmp byte[cho],32h

je b

jmp exit

a: call Succe\_addition

jmp \_start

b: call Add\_shift

jmp \_start

exit:

mov rax,60 mov rdi,0 syscall

convert: ;; ASCII to Hex conversion xor rbx,rbx xor rcx,rcx

xor rax,rax

mov rcx,02 mov rsi,num up1: rol bl,04

mov al,[rsi] cmp al,39h jbe p1 sub al,07h jmp p2

p1: sub al,30h p2: add bl,al inc rsi

loop up1

ret

display: ;; Hex to ASCII conversion

mov rcx,4 mov rdi,result dup1: rol bx,4 mov al,bl and al,0fh cmp al,09h jbe p3 add al,07h jmp p4 p3: add al,30h p4:mov [rdi],al inc rdi

loop dup1

mov rax,1 mov rdi,1 mov rsi,result mov rdx,4 syscall

ret

Succe\_addition:

mov rax,1 mov rdi,1 mov rsi,msg mov rdx,msg\_len syscall

mov rax,0 mov rdi,0 mov rsi,num mov rdx,3

syscall

call convert mov [num1],bl

mov rax,1 mov rdi,1 mov rsi,msg mov rdx,msg\_len syscall

mov rax,0 mov rdi,0 mov rsi,num mov rdx,3 syscall

call convert xor rcx,rcx xor rax,rax mov rax,[num1]

repet: add rcx,rax dec bl jnz repet

mov [result],rcx

mov rax,1 mov rdi,1 mov rsi,res mov rdx,res\_len syscall

mov rbx,[result]

call display ret Add\_shift:

mov rax,1 mov rdi,1 mov rsi,msg mov rdx,msg\_len syscall

mov rax,0 mov rdi,0 mov rsi,num mov rdx,3 syscall

call convert mov [num1],bl

mov rax,1 mov rdi,1 mov rsi,msg

mov rdx,msg\_len syscall

mov rax,0 mov rdi,0 mov rsi,num mov rdx,3 syscall

call convert

mov [num],bl

xor rbx,rbx xor rcx,rcx xor rdx,rdx xor rax,rax mov dl,08 mov al,[num1]

mov bl,[num]

p11:

shr bx,01 jnc p add cx,ax p: shl ax,01 dec dl

jnz p11

mov [result],rcx

mov rax,1 mov rdi,1 mov rsi,res mov rdx,res\_len

syscall

;dispmsg res,res\_len

mov rbx,[result]

call display

ret

OUTPUT:

;;Enter your Choice:

;;1.Successive Addition

;;2.Add and Shift method

;;3.Exit

;;1

;;Enter two digit Number::

;;02

;;Enter two digit Number::

;;02

;;Multiplication of elements is::0004Enter your Choice:

;;1.Successive Addition

;;2.Add and Shift method

;;3.Exit

;;2

;;Enter two digit Number::

;;03

;;Enter two digit Number::

;;03

;;Multiplication of elements is::0009Enter your Choice:

;;1.Successive Addition

;;2.Add and Shift method

;;3.Exit

;;3

**CONCLUSION:** In this practical session we learnt how to perform multiplication of two 8-bit hexadecimal numbers using successive addition and add and shift method.

**EXPERIMENT NO. 10**

**NAME:** Write x86 ALP to find the factorial of a given integer number on a command line by using recursion. Explicit stack manipulation is expected in the code.

**AIM:** Write x86 ALP to find the factorial of a given integer number on a command line by using recursion. Explicit stack manipulation is expected in the code.

**OBJECTIVES:**

* To understand assembly language programming instruction set.
* To understand different assembler directives with example.
* To apply instruction set for implementing X86/64 bit assembly language programs

**ENVIRONMENT:**

* Operating System: 64-bit Open source Linux or its derivative.
* Programming Tools: Preferably using Linux equivalent or MASM/TASM/NASM/FASM.  Text Editor: geditor

**THEORY:**

A recursive procedure is one that calls itself. There are two kind of recursion: direct and indirect. In direct recursion, the procedure calls itself and in indirect recursion, the first procedure calls a second procedure, which in turn calls the first procedure.

Recursion could be observed in numerous mathematical algorithms. For example, consider the case of calculating the factorial of a number. Factorial of a number is given by the equation −

Fact (n) = n \* fact (n-1) for n > 0

For example: factorial of 5 is 1 x 2 x 3 x 4 x 5 = 5 x factorial of 4 and this can be a good example of showing a recursive procedure. Every recursive algorithm must have an ending condition, i.e., the recursive calling of the program should be stopped when a condition is fulfilled. In the case of factorial algorithm, the end condition is reached when n is 0.

Recursion occurs when a procedure calls itself. The following for example is a recursive procedure:

Recursive proc callRecursive ret

Recursive endp

Of course the CPU will never execute the ret instruction at the end of this procedure. Upon entry into Recursive this procedure will immediately call itself again and control will never pass to the ret instruction. In this particular case run away recursion results in an infinite loop.

In many respects recursion is very similar to iteration (that is the repetitive execution of a loop).

The following code also produces an infinite loop:

Recursive proc

jmp Recursive

ret

Recursive endp

There is however one major difference between these two implementations. The former version of Recursive pushes a return address onto the stack with each invocation of the subroutine. This does not happen in the example immediately above (since the jmp instruction does not affect the stack).

Like a looping structure recursion requires a termination condition in order to stop infinite recursion. Recursive could be rewritten with a termination condition as follows:

Recursive proc dec ax jzQuitRecurse call Recursive

QuitRecurse: ret

Recursiveendp

This modification to the routine causes Recursive to call itself the number of times appearing in the ax register. On each call Recursive decrements the ax register by one and calls itself again. Eventually Recursive decrements ax to zero and returns. Once this happens the CPU executes a string of ret instructions until control returns to the original call to Recursive.

So far however there hasn't been a real need for recursion. After all you could efficiently code this procedure as follows:

Recursive proc RepeatAgain: dec ax jnzRepeatAgain

ret

Recursive endp

Both examples would repeat the body of the procedure the number of times passed in the ax register. As it turns out there are only a few recursive algorithms that you cannot implement in an iterative fashion. However many recursively implemented algorithms are more efficient than their iterative counterparts and most of the time the recursive form of the algorithm is much easier to understand.

**ALGORITHM:**

Step1: Start

Step2: Accept the number from user

Step3: Convert that number into Hexadecimal (ascii to hex)

Step4: Compare accepted number with 1. If it is equal to 1 go to step 5 else push the number on stack and decrement the number and go to step 4

Step5: pop the content of the stack and multiply with number

Step6: Repeat step until stack becomes empty

Step7: Convert number from Hex to ASCII

Step8: Print the number

Step9: Stop

**PROGRAM:**

**OUTPUT:**

**CONCLUSION:** In this practical session we learnt how to find the factorial of a given integer number on a command line by using recursion.